Programming Assignment Report

Program 4

CPU scheduling

Abstract

The purpose of this lab was to learn and understand how to use Round Robin and First Come First Served CPU scheduling in Java. In the lab, a CPU scheduler was implemented to manage multiple processes and compute key performance metrics such as CPU Utilization, Throughput, Average Waiting Time, and Average Turnaround Time. The output from both algorithms was compared to evaluate their strengths and weaknesses in terms of fairness, efficiency, and responsiveness. Through this lab, the differences in how FCFS and RR handle process execution was shown.

Introduction

3.1 Problem Statement

Understand how FCFS and RR work and program in java using the provided code as a base.

3.2 Objective

To understand how different CPU schedulers can have an effect on the runtime of different processes.

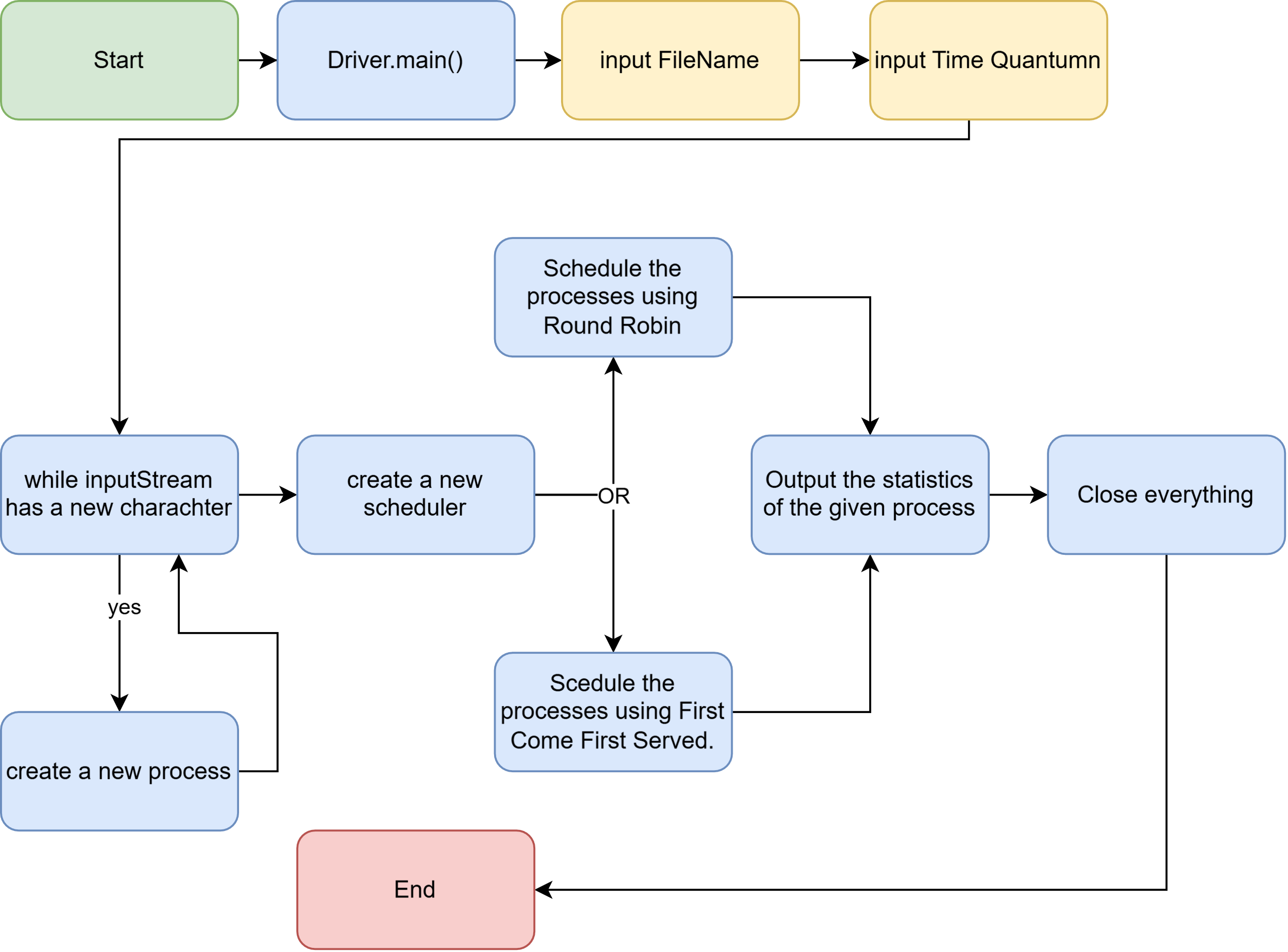
3.3 Background Information

The concepts used within the lab relate to CPU Scheduling, more specifically First Come First Served and Round Robin.

Materials and Methods

4.1 Tools and Technologies

For the assignment we used several different software types to help complete the lab. For the main part of the assignment, we used a Virtual ubuntu machine to run our Linux environment. Within the environment we used the terminal to access all the operating systems programs. We used the nano text editor to write Java code to then compile using javac compiler. We then were given multiple starting files that had missing portions to fill out. Once filled out we would run it with each separate scheduling method by uncommenting the specific line.

4.2 Program Design  


4.3 Code Implementation

Using the provided files I implemented the missing code for statistic calculations, round robin implementation, and first come first served. When testing if it worked, I verified that it first compiled correctly, then tested using the example input and the input csv file.

Results

5.1 Test Cases

I ran each of the given scripts to test the outputs of each file. The outputs are below.

5.2 Output

Discussion

6.1 Post-Lab Questions

N/A

6.2 Challenges Encountered and Improvements

A challenge I faced was attempting to run the scripts the first time, I had forgotten to allow them to be executed. This was solved by running chmod on the files, changing them to executable. a possible improvement would be to add more arguments to the racer class such as thread count.

Conclusion

I understand how the GetOpt command works and how to use it in Java programs, I also understand how Threads work when making a multithreaded program. The output of the racer program does match the way a multithreaded program should act by showing different start and stop times, but not in ascending order.

Screenshots

N/a

Source Code:

Racer.java

|  |
| --- |
| class Racer implements Runnable { // input the interface name  private String name;  private int M = 0; // these fields are shared by both  private volatile long sum = 0; // threads since there is one object  public Racer(String name, int M) {  this.name = name;  this.M = M;  System.out.println("age()=" + Scheduler.age() + ", "  + name + " is alive, M=" + M);  }  private long fn(long j, int k) {  long total = j;  for (int i = 1; i <= k; i++)  total += (2 \* i - 1) \* (2 \* i - 1);  return total;  }  public void run() { // input the abstrac function name  System.out.println("age()=" + Scheduler.age() + ", "  + name + " is running");  for (int m = 1; m <= M; m++) {  /\*  \* "N = N + 1" type lost update (race condition) in following line  \*/  sum = fn(sum, m);  }  System.out.println("age()=" + Scheduler.age() + ", "  + name + " is done, sum = " + sum);  }  }  class RaceTwoThreads {  private static int M = 10;  private final static int numRacers = 2; // fill in a number that how many threads you want to run  public static void main(String[] args) {  // parse command line arguments, if any, to override defaults  GetOpt go = new GetOpt(args, "UtM:");  go.optErr = true;  String usage = "Usage: -t -M m";  int ch = -1;  boolean timeSlicingEnsured = false;  while ((ch = go.getopt()) != go.optEOF) {  if ((char) ch == 'U') {  System.out.println(usage);  System.exit(0);  } else if ((char) ch == 'M')  M = go.processArg(go.optArgGet(), M);  else {  System.err.println(usage);  System.exit(1);  }  }  System.out.println("RaceTwoThreads: M=" + M + ", timeSlicingEnsured="  + timeSlicingEnsured);  // start the two threads, both in the same object  // so they share one instance of its variable sum  Racer r = new Racer("Racer", M); // create a thread function pobject  Thread[] threads = new Thread[numRacers]; // create an array of threads  for (int i = 0; i < numRacers; i++)  threads[i] = new Thread(r, "RacerThread-" + i); // create a racer thread  for (int i = 0; i < numRacers; i++) {  threads[i].start(); // start each thread  }  System.out.println("age()=" + Scheduler.age() +  ", all Racer threads started");  // wait for them to finish if not forced consecutive  try {  for (int i = 0; i < numRacers; i++) {  threads[i].join(); // synchronize threads using join()  }  } catch (InterruptedException e) {  System.err.println("interrupted out of join");  }  // correct race-free final value of sum is 2\*220 = 440 for M of 10  // and 2\*1335334000 = 2670668000 for M of 2000 (so `long sum' needed)  System.out.println("RaceTwoThreads done");  System.exit(0);  }  } |

Appendix

n/a